Rule-based Model Extraction from Source Code

Rui Correia[1,2], Carlos Matos[1,2], Mohammad El-Ramly[1], and Reiko Heckel[1]

[1] Department of Computer Science, University of Leicester, U.K.
{rmc20, cmm22, mer14, reiko}@mcs.le.ac.uk
[2] ATX Software, Portugal

Abstract. In the context of an approach for reengineering legacy software systems at the architectural level, we present in this paper a reverse engineering methodology that uses a model defined as a type graph to represent source-code subject to a code categorization process. Two alternative methods for referencing the source code are discussed: native vs. graphical. To represent the code, the native representation uses the abstract syntax tree while the graphical uses a programming language metamodel. Two options regarding the way that the graph can relate to the source code reference model are also considered: association model vs. direct link. The extraction of the program representation, complying to the type graph, is based on rules that categorize source code according to its purpose. The techniques to address this process, such as the code categorization rules, are shown together with examples.

1 Introduction

In this paper, we present a reverse engineering process that addresses the needs of a methodology for architectural transformation that is being developed in the context of Leg2Net project [1]. This reengineering methodology's goal is to define a process through which, given a legacy system with some known architecture and the requirement to transform it into another known target architecture, we can extract code fragments that correspond to the different source architectural elements and use them to build the target architecture. This approach aims not to limit itself to transformations between specific source and target architectures or programming languages. Our objective on a second project, SENSORIA [2], is to develop a methodology and tools for transformations that target Service-Oriented Architectures. This paper will focus on the extraction of a source code representation. This process is performed according to a set of code categorization rules and the model achieved will comply to a metamodel defined as a type graph. This model representation, which is called Program Representation Graph (PRG), needs to keep traceability to the source code for reasons concerned with the reengineering methodology. In the paper we suggest two different approaches to reference the source code, one based on the Abstract Syntax Tree (AST) and another one based on a metamodel of the source programming language. The

Dagstuhl Seminar Proceedings 06302
Aspects For Legacy Applications
http://drops.dagstuhl.de/opus/volltexte/2007/881

relation between the program representation graph and the model used to reference the source code can be a simple link between the two representations or an association model.

The rest of this paper is organized as follows:

- Section 2 gives an overview of the full reengineering methodology;
- Section 3 and its subsections present the reverse engineering methodology including the models involved and the rule-based extraction technique;
- Section 4 discusses related work;
- Section 5 presents the conclusions and future work.

2 Reengineering Methodology

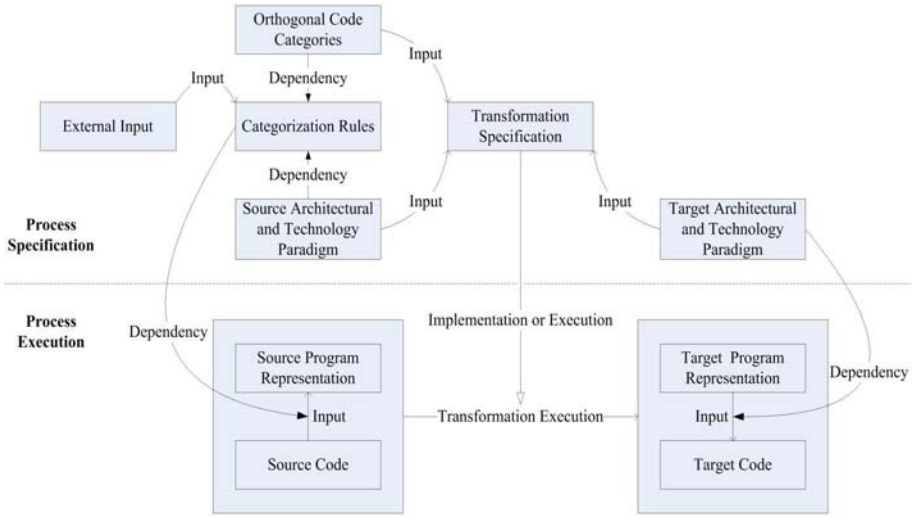

Fig. 1: General Approach

The full approach, as shown in Figure 1, is a reengineering solution composed of three steps derived from the Horseshoe Model [3]:

1. Reverse engineering, in order to achieve a representation of the source code through its categorization and structuring;
2. Transformation techniques to achieve a representation of the target code from the source code representation previously accomplished;
3. Generation of the target code based on the combination of the source code and the target code representation previously achieved.

In this paper we focus on the reverse engineering part of the methodology, where the source code is divided into blocks that are categorized according to

their purpose, for example User Interface Action or Data Definition, using the information contained in the Categorization Rules (Figure 1). The output of this process is the Source Program Representation which complies to the model specified in the Architectural and Technology Paradigm.

The code categories used for the classification are closely related with the semantics of the source code. They are independent of the architectural and technology paradigm to which the program belongs. They can be divided in two types:

- Composed by a concern and a role
- Connectors representing links between concerns

Concerns are conceptual classifications of code that are assigned regarding its goal. A non-exclusive list of identified concerns is:

- User Interface (UI)
- Business Logic (BL)
- Data

Roles are classifications of code according to its execution processing. Some of the identified roles are:

- Definition
- Action
- Validation

Concerns and roles are transversal concepts. A code category of this type can consist of any combination of the two, for example: Business Logic Action or User Interface Validation.

The connectors are one-way (non-commutative) links between different concerns and include:

- Data storage/retrieval
- Network/communication
- Control: UI to BL
- Control: BL to UI
- Control: BL to Data
- Control: Data to BL

Regarding the rest of the methodology, the transformation techniques are based in graph transformations, some of which are refactoring [4] primitives. The target code can be reached by code generation techniques using the source code and the transformation output. However, this is not the focus of this article.

3 Categorization Process

The categorization process consists of the division of the source code into blocks that are categorized according to their purpose. The source code is submitted to a set of categorization rules which result is a program representation expressed using an instance of the model defined by the Architectural and Technology Paradigm (Figure 1). The next subsections will detail the elements involved in this process and examples will be provided.

3.1 Architectural and Technology Paradigm

The Architectural and Technology Paradigm contains a model for the programming language paradigm, which is specified by means of a type graph. This model is called the Program Representation Graph (PRG). In Figure 2, it is possible to see a simplified model for OO. This is an extension of the type graph presented by Tom Mens et al in [5]. It was necessary to extend that model in order to introduce classification attributes and the notion of code block. This had to be done because we have the need of lower granularity level than the method for the process of code classification. Since it is necessary to keep traceability to

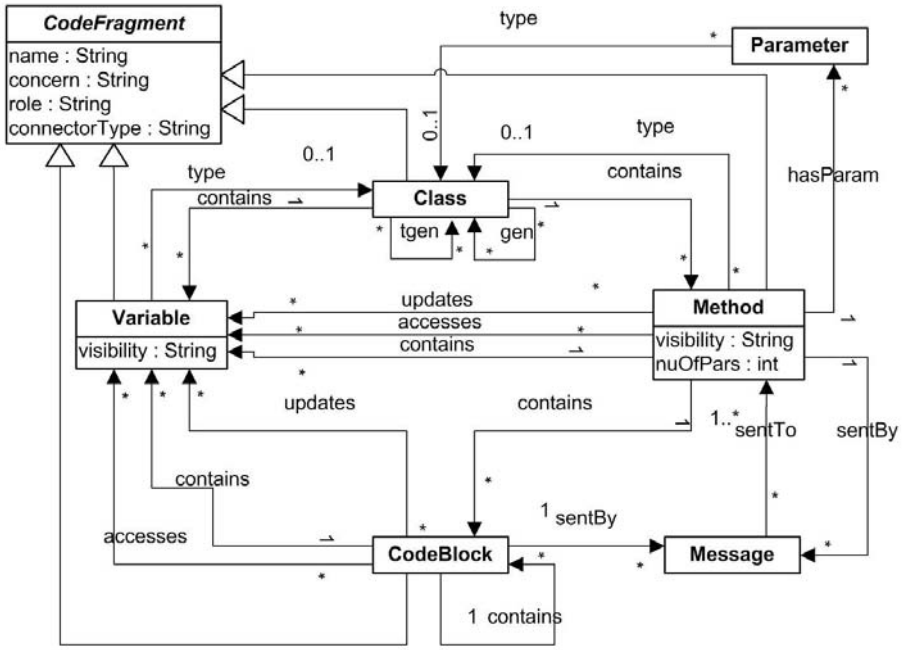

Fig. 2: Program Representation Graph (type graph) for the OO paradigm (simplified)

the source code for reasons concerned with the full reengineering methodology, a method to associate it to the PRG has to be considered. Given that we want the methodology to be as programming language independent as possible we will not link the PRG directly to the source code. Hence, we are studying two solutions to address this issue:

1. A native representation based on an Abstract Syntax Tree (AST)
2. A graphical representation using a metamodel for the programming language

ASTs are very well known representations of source code and allow the referencing that we need. Programming language metamodels are perhaps more difficult

to define but are also a solution to take into account. For Java, we can obtain the AST using an Eclipse API from Java Development Tools (JDT) [6]. As for the metamodel, an Eclipse Modeling Framework (EMF [7]) based metamodel for Java called Java EMF Model (JEM) is part of the Eclipse's Visual Editor project, but is still in its early development stages [8].

A different issue is the type of relation between the PRG and the model used to reference the source code. We are considering the following alternatives:

1. An extension to the type graph (PRG) by adding an attribute to the abstract class *CodeFragment* that uniquely identifies the equivalent element in the source code reference model
2. The use of an association model that defines the mapping between elements of the PRG and corresponding elements of the source code reference model

The first alternative is a straight-forward solution that has a tight relation between both models. This is a drawback because with this option the PRG will depend on the chosen source code reference model. Its main advantage is the simplicity of implementation where no extra models are necessary. To use this solution the only extension necessary is an extra attribute in the abstract class *CodeFragment* of the type graph. The second alternative has greater flexibility because it uses an association model to link the source code reference model and the PRG. This is more flexible than the first alternative because it allows the PRG to be independent of the chosen source code reference model. However, this flexibility adds complexity since a model for the association is required. By considering alternatives for the source code reference model and the relation model used to link it to with the PRG, we get four possible options. We present two of the alternatives in Figures 3 and 4. Figure 3 shows an example using JEM as the source code reference model and an association model to the PRG for the

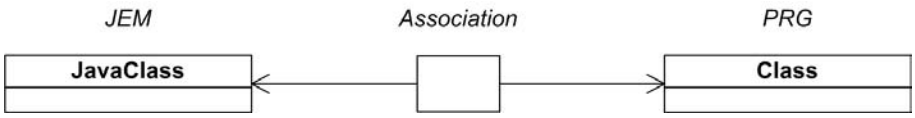

Fig. 3: Example using JEM as the source code reference model and an association model to the PRG

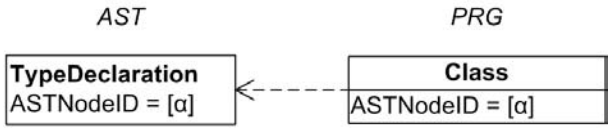

Fig. 4: Example using the AST from Eclipse JDT and an extra attribute in the PRG to act as link (the dashed arrow is only for illustrative purposes)

element *Class*. Figure 4 presents an example using the AST from Eclipse JDT and the PRG extended with an attribute to reference it.

For simplicity reasons, in the rest of the paper only one of the possible four solutions will be used: the AST as source code reference and the extended PRG to provide the link between them (as in Figure 4).

3.2 Program Representation

The Program Representation is an abstraction of the code. It is achieved by mapping the code into the categories defined in the Orthogonal Code Categories, forming a structure that stores that information together with control / data dependencies. We are using a graph notation that is an instance of the type graph previously defined and shown in Figure 2, the PRG, where the code is categorized and its dependencies are defined. An example can be seen in Figure 5.

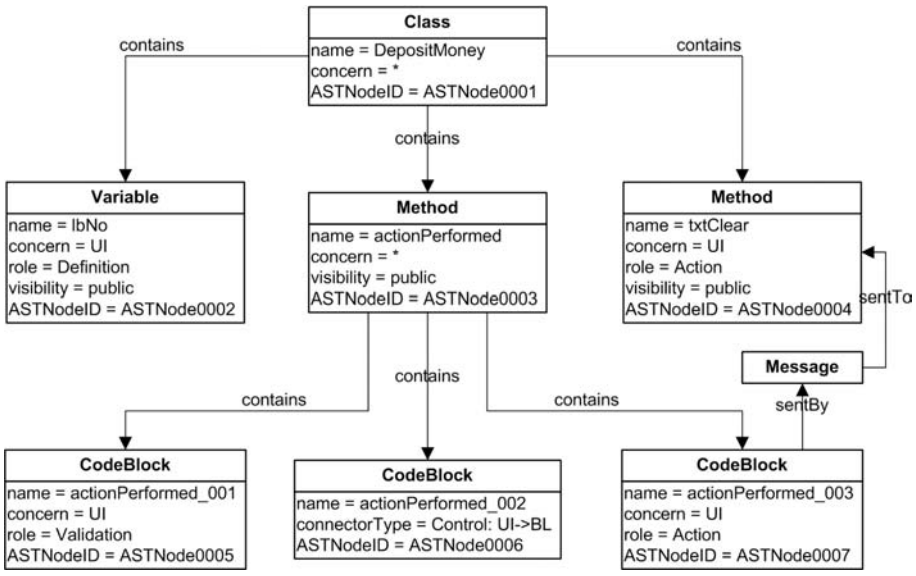

Fig. 5: Graph representing a subset of a Java sample application

Figure 5 is an instance of the extended type graph with the *ASTNodeID* attribute used to reference the corresponding AST node. This graph is obtained from the AST presented in Figure 6b. The corresponding source code can be seen in Figure 6a.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DepositMoney extends JInternalFrame
         implements ActionListener { // ASTNode0001
  private JLabel lbNo /* ASTNode0002 */, lbName,
                 lbDate, lbDeposit; // ASTNode0010
  private JTextField txtNo, txtName, txtDeposit;
  private JButton btnSave, btnCancel;
  // (...)
  public void actionPerformed(ActionEvent ae) {
                                    // ASTNode0003
    Object obj = ae.getSource();
    if(obj == btnSave) { // ASTNode0005
      if(txtNo.getText().equals("")) {
        JOptionPane.showMessageDialog(this,
                "Customer id", "Empty",
                JOptionPane.PLAIN_MESSAGE);
        txtNo.requestFocus();
      }
      else {
        if(txtDeposit.getText().equals("")) {
          JOptionPane.showMessageDialog(this,
                  "Amount", "Empty",
                  JOptionPane.PLAIN_MESSAGE);
          txtDeposit.requestFocus();
        }
        else { // ASTNode0006
          editRec();
        }
      }
    }
    if(obj == btnCancel) { // ASTNode0007
      txtClear();
      setVisible(false);
      dispose();
    }
  }
  //(...)
  void txtClear() { // ASTNode0004
    txtNo.setText("");
    txtName.setText("");
    txtDeposit.setText("");
    txtNo.requestFocus();
  }
  //(...)
  public void editRec () {
    //(...)
  }
  //(...)
}
```

(a) Example source code

```
PACKAGE: null
IMPORTS(3)
TYPES(1)
  TypeDeclaration
    ASTNode0001
    type binding: DepositMoney
    BODY_DECLARATIONS(6)
      FieldDeclaration
        ASTNode0010
        TYPE
          SimpleType
            type binding:
            javax.swing.JLabel
        FRAGMENTS(4)
          VariableDeclarationFragment
            ASTNode0002
            variable binding:
            DepositMoney.lbNo
      (...)
      MethodDeclaration
        ASTNode0003
        method binding:
        DepositMoney.actionPerformed()
        BODY
          IfStatement
            ASTNode0005
            IfStatement
              EXPRESSION
              THEN_STATEMENT
              ELSE_STATEMENT
                IfStatement
                  EXPRESSION
                  THEN_STATEMENT
                  ELSE_STATEMENT
                    ASTNode0006
            IfStatement
              ASTNode0007
      MethodDeclaration
        ASTNode0004
        method binding:
        DepositMoney.txtClear()
      (...)
```

(b) Example AST

Fig. 6: Source code and AST extracts from a Java sample application.

3.3 Categorization Rules

The rules that are used in the categorization process are based on the source code characteristics and are applied over the AST. The definition of these rules

is not closed but some of them were already obtained. The following examples can be given in an informal way:

1. Statements that consist of variable/attribute declarations for a type that is known to belong to a certain concern, will be categorized as belonging to the same concern and having the role Definition.
   Example: the Java statement '`private JLabel lbNo;`' is categorized as UI Definition because it is known that JLabel belongs to the UI concern;
2. Attributions to variables/attributes that are known to belong to a certain concern and whose right hand side only includes the use of elements (e.g. variables or method invocations) that belong to the same concern, will have that concern and the role Definition.
   Example: the Java statement '`lbNo = new JLabel ("Account No:");`' is categorized as UI Definition because it is known that the attribute lbNo and the JLabel method/constructor invocation belongs to the UI concern;
3. Variables/attributes/parameters definition/attribution that are used to store values directly from Data Action methods/functions belong to the Data Action category.
   Example: the Java statement '`records[rows][i] = dis.readUTF();`' belongs to the Data Action because the readUTF operation is know to belong to that category.

The rules will have to be applied in multiple-pass. The reason for this is the fact that the application of a rule can enable the application of another. An example for this need can be given using rule number 2: if a method invocation that exists in the right hand side of the attribution is not yet categorized, the rule will not be applied. However, after some other rule categorizes the method, rule number 2 can be applied. The implementation of an engine that supports the categorization process will have to contain stop conditions.

The categorization rules are defined formally with left hand side (LHS) and right hand side expressions (RHS). The LHS consists in the prerequisites that must be satisfied in order to apply the rule. The RHS is the result of the rule application. The expressions are based on the architectural and technology paradigm models. Both in the LHS and RHS there can be elements from the AST and the type graph. An example of rule definition and the result of applying it is presented in the next subsection.

3.4 Example

The rule number 1 (introduced in the previous subsection) can be represented as shown in Figure 7 for the declaration of attributes of type JLabel. The LHS expresses the type of nodes that are matched by the rule: all AST nodes of type *FieldDeclaration* that have the base type *SimpleType* and type name *JLabel*. In the AST, a field declaration statement has a list of children; each element corresponds to an attribute being declared in the statement. These are called declaration fragments. The RHS shows that the rule application results in the

creation of a node of type *Variable* in the instance graph for each of the declaration fragments. This node will have the attributes *concern* and *role* with the values "UI" and "Definition", respectively. The attribute *name* of the new graph nodes is the same of the declaration fragments in the AST. The relation between the graph node and the AST is made via the *ASTNodeID* attribute.

An example of the application of this rule can be seen by relating Figures 5 and 6b. The LHS of rule number 1 is matched in the AST (Figure 6b) with the variables instantiated as follows: $[\alpha]$ = "ASTNode0010", $[\beta]$ = "ASTNode0002" and [X] = "lbNo". The result of applying the rule, as stated by the RHS, is the creation of the *Variable* element in the PRG with *name* = "lbNo", *concern* = "UI", *role* = "Definition" and *ASTNodeID* = "ASTNode0002" (Figure 5). This last attribute of the new graph element is the one that specifies its relation to the AST.

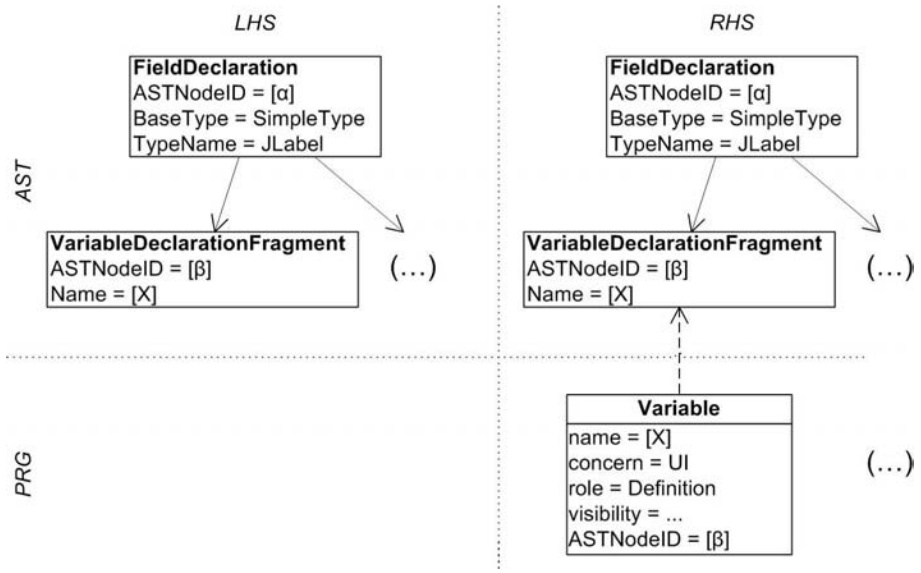

Fig. 7: Categorization rule 1 (Variable/attribute declaration of known type) example definition for the JLabel type. (The dashed arrow is used in this figure only to make the relation between the graph and the AST more visually explicit.) Note that the text between square brackets represents variables

4 Related Work

There is a lot of work in the area of source code representation. Here we brief a few example works to show how this issue is dealt within different contexts.

The Dagstuhl Middle Model (DMM) [9] was developed to solve interoperability issues of reverse engineering tools. Like our approach, it keeps traceability

to the source code. The DMM is composed by sub-hierarchies that include an abstract view of the program and a source code model. The chosen way to relate these two is via a direct link. The Fujaba (From UML to Java And Back Again) tool suite [10] uses design pattern [11] recognition. The source code representation used for that process is based on an Abstract Syntax Graph (ASG). Another representation is put forward with the Columbus Schema for C++ [12]. Here an AST conforming to the C++ model/schema is built, and a higher level semantic information is derived from types. The work of Ramalingam et al, from IBM research, addresses the reverse engineering of OO data models from programs written in weakly-typed languages like Cobol. In their work, the links between the model and the code are represented in a reference table. This table establishes the link between each model element and the line of code having no intermediate representation [13].

The ARTISAn framework, described by Jakobac, Egyed and Medvidovic in [14], like our approach, categorizes source code. The code categories used are: "processing", "data" and "communication". The approach differs from ours in several aspects. Firstly, the goal of the framework is program understanding and not the creation of a representation that is aimed to be used as input for the transformation part of a reengineering methodology. Another important difference is that in ARTISAn the categorization process (called "labeling") is based in clues that result in the categorization of classes only. In our approach we need, and support, the method and code block granularity levels.

5 Discussion, Conclusions and Future Work

In this paper we presented a specific program representation model for reengineering purposes. Namely, it is necessary that this model keeps source code traceability and addresses a code categorization process that classifies code blocks according to their purpose. In this context, we presented a model for the OO paradigm based on an attributed type graph and provided alternate ways for source code reference models. Alternatives were also given to the way that the program representation graph should be linked to the source code reference model. We discussed the advantages and disadvantages of these alternatives and provided examples for one of them. The rule-based technique used to obtain an instance of the program representation was also presented along a sample of the code categorization rules. The next steps in the development of this reverse engineering methodology are:

- Researching existing programming language metamodels for Java, starting with JEM, and then also similar work being done for other languages
- Evaluating empirically the best approach as source code reference model and the way to link it to the PRG
- Defining a set of code categorization rules that can be applied in a real-world scenario
- Automating the code categorization process

Acknowledgments

R. Correia and C. Matos are Marie-Curie fellows seconded to the University of Leicester as part of the Transfer of Knowledge, Industry Academia Partnership Leg2Net (MTK1-CT-2004-003169). This work has also been supported by the IST-FET IP SENSORIA (IST-2005-16004).

We would like to thank José Luiz Fiadeiro (University of Leicester), Georgios Koutsoukos and Luís Andrade (both from ATX Software) for their contribution in the reengineering methodology development.

References

1. Leg2Net: From legacy systems to services in the net. (http://www.cs.le.ac.uk/SoftSD/Leg2Net/)
2. SENSORIA: Software engineering for service-oriented overlay computers. (http://sensoria.fast.de/)
3. Kazman, R., Woods, S.G., Carrière, S.J.: Requirements for integrating software architecture and reengineering models: CORUM II. In: Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98), Washington, DC, USA, IEEE Computer Society (1998) 154–163
4. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Boston, MA, USA (1999)
5. Mens, T., Taentzer, G., Runge, O.: Analyzing refactoring dependencies using graph transformation. Software Systems Modeling (SoSyM), to appear (2006)
6. Eclipse: JDT - AST. (http://help.eclipse.org/help30/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/AST.html)
7. Eclipse: Eclipse modeling framework. (http://www.eclipse.org/emf/)
8. Eclipse: Visual editor project. (http://www.eclipse.org/vep)
9. Lethbridge, T.C., Plödereder, E., Tichelaar, S., Riva, C., Linos, P., Marchenko, S.: The Dagstuhl Middle Model (DMM). (http://www.ece.queensu.ca/hpages/courses/elec875/pdf/DMMDescriptionV0006.pdf)
10. Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J.: Towards pattern-based design recovery. In: Proceedings of the Twenty Fourth International Conference on Software Engineering (ICSE'02), Orlando, Florida, USA, ACM Press (2002) 338–348
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
12. Ferenc, R., Árpád Beszédes: Data exchange with the columbus schema for c++. In: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR'02), Washington, DC, USA, IEEE Computer Society (2002) 59–66
13. Ramalingam, G., Komondoor, R., Field, J., Sinha, S.: Semantics-based reverse engineering of object-oriented data models. In: Proceeding of the Twenty Eighth International Conference on Software Engineering (ICSE'06), New York, NY, USA, ACM Press (2006) 192–201
14. Jakobac, V., Egyed, A., Medvidovic, N.: Improving system understanding via interactive, tailorable, source code analysis. In: Fundamental Approaches to Software Engineering (FASE'05), Springer Berlin / Heidelberg (2005) 253–268